



Sheep in wolf's clothing: Implementation models for data-flow multi-threaded software

Keryan Didier, Albert Cohen, Adrien Gauffriau, Amaury Graillat, Dumitru Potop-Butucaru

► To cite this version:

Keryan Didier, Albert Cohen, Adrien Gauffriau, Amaury Graillat, Dumitru Potop-Butucaru. Sheep in wolf's clothing: Implementation models for data-flow multi-threaded software. [Research Report] RR-9057, Inria Paris. 2017, pp.31. hal-01509314

HAL Id: hal-01509314

<https://hal.inria.fr/hal-01509314>

Submitted on 16 Apr 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Sheep in wolf's clothing: Implementation models for data-flow multi-threaded software

Keryan Didier, Albert Cohen, Adrien Gauffriau, Amaury Graillat,
Dumitru Potop-Butucaru

**RESEARCH
REPORT**

N° 9057

April 7, 2017

Project-Teams AOSTE,PARKAS

ISSN 0249-6399

ISBN INRIA/RR--9057--FR+ENG



Sheep in wolf's clothing: Implementation models for data-flow multi-threaded software

Keryan Didier, Albert Cohen, Adrien Gauffriau, Amaury Graillat, Dumitru Potop-Butucaru

Project-Teams AOSTE,PARKAS

Research Report n° 9057 — April 7, 2017 — 28 pages

Abstract: Concurrent programming is notoriously difficult, especially in constrained embedded contexts. Threads, in particular, are wildly nondeterministic as a model of computation, and difficult to analyze in the general case. Fortunately, it is often the case that multi-threaded, semaphore-synchronized embedded software implements high-level functional specifications written in a deterministic data-flow language such as Scade or (safe subsets of) Simulink. We claim that in this case the implementation process should build not just the multi-threaded C code, but (first and foremost) a richer model exposing the dataflow organization of the computations performed by the implementation. From this model, the C code is extracted through selective pretty-printing, while knowledge of the data-flow organization facilitates analysis. We propose a language for describing such implementation models that expose the data-flow behavior (the sheep) hiding under the form of a multi-threaded program (the wolf). The language allows the representation of efficient implementations featuring pipelined scheduling and optimized memory allocation and synchronization. We show applicability on a large-scale industrial avionics case study and on a commercial many-core.

Key-words: synchronous languages, Kahn process networks, execution platform, semantic preservation, implementation model, multi-thread, parallelisme, Lustre, Scade

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Brebis déguisées en loups : Modèles d'implémentation pour systèmes parallèles flots de données

Résumé : La programmation concurrente est une discipline difficile, particulièrement dans un contexte de systèmes embarqués. Les threads, en particulier, sont un modèle de calcul non-déterministe et difficile à analyser dans le cas général. Heureusement, les logiciels embarqués multi-threadés synchronisés par sémaphores sont souvent des implémentations de spécifications fonctionnelles de haut niveau écrites dans un langage flot-de-données déterministe comme Scade ou (un sous-ensemble sûr de) Simulink. Dans ce cas, le processus d'implémentation devrait, non seulement construire le code C multi-threadé de l'implémentation, mais avant tout un modèle plus riche exposant l'organisation du flot-de-données des calculs effectués par le code. De ce modèle, le code C peut être extrait par du simple *pretty printing*. En même temps, la structure flot-de-donnée facilite l'analyse. Nous proposons un langage pour la description de tels modèles d'implémentations qui exposent le comportement flot-de-donnée (la brebis) déguisé en un programme multi-threadé (le loup). Ce langage permet une représentation d'implémentations efficaces avec ordonnancement pipeliné et allocation mémoire et synchronisations optimisées. Nous montrons son application sur un cas d'étude de l'industrie aéronautique et sur une plateforme many-coeurs commerciale.

Mots-clés : langages synchrones, réseaux de Kahn, plateforme d'exécution, préservation sémantique, modèle d'implémentation, multi-thread, parellélisme, Lustre, Scade

Contents

1	Introduction	4
1.1	Motivating example.	4
1.2	Related work and originality	7
2	Modifications to Lustre	7
2.1	Synchronous semantics principles	11
2.2	Kahnian asynchronous interpretation	11
2.2.1	Preliminary notations	11
2.2.2	Program state	12
2.2.3	Semantic rules and semantics preservation	13
2.2.4	Properties ensuring implementability	14
3	Implementation modeling	14
3.1	Target architectures	14
3.1.1	API and ABI	15
3.2	Mapping information	15
3.3	Platform semantics	18
3.3.1	State representation	18
3.3.2	Semantic rules	20
3.3.3	Correctness and semantics preservation	21
4	Experimental evaluation	21
5	Conclusion	22
A	Synchronous semantics of InteLus	23
A.1	State representation	23
A.2	Notations	23
A.3	Structural operational semantics rules	23
A.4	Clocks	25
A.5	Equation guards	27

1 Introduction

Mastering concurrency is difficult, and yet hardware design resolutely moves towards increasingly massive parallelism based on the use of chip-multiprocessors. Threads [6] are one of the major programming paradigms for such multi- and many-core systems. They arguably provide the best portability and the finest control over resource allocation, which are both essential in the design of embedded applications that need to get the best guaranteed performance out of resource-constrained hardware.

But such expressiveness comes at a price. As a model of computation, threads are wildly non-deterministic and non-compositional [9], making both programming and formal analysis [13] difficult. This explains why multi-threaded software is often bug-ridden even in the context of critical systems [8].

But there are also good news: in many industrial contexts (avionics, automotive, *etc.*) the use of threads is tightly controlled. We consider in this paper the particular case where the functional specification of the system is done in a synchronous dataflow language such as Simulink¹ or Scade². In this case, multi-threaded implementations have particular structure and properties: The number of threads is fixed, each one implementing a recurrent task obtained by sequencing a fixed set of dataflow blocks (or parts thereof, obtained by parallelization).

Such hypotheses, when taken individually, largely facilitate the formal analysis of multi-threaded systems. But *in many cases, the multi-threaded implementation preserves a fundamentally dataflow structure*, with specific rules on the way platform resources (shared memory, semaphores) are used. When this happens, the implementation is best represented as a dataflow synchronous program whose elements are mapped on the platform resources. Ensuring the correctness of such an implementation consists in ensuring that:

1. The dataflow program (without the mapping) implements the semantics of the functional specification. This analysis can be performed inside the dataflow model.
2. Once the mapping of program elements onto the platform resources³ is performed, the execution of the platform (under platform semantics) implements the behavior of the dataflow program.

Together, the dataflow program and the mapping information form an *implementation model*. This model is strictly richer than the multi-threaded C code, which can be obtained through a pretty-printing of model parts. Exposing the internal data-flow structure of the implementation facilitates defining and establishing correctness, *e.g.* the correctness of the synchronization or memory coherence protocols synthesized during the implementation process. All analyses can be realized using efficient tools specific to the synchronous model. Finally, if manual inspection of the C multi-threaded code is required, such a representation can be used to enforce strict code structuring rules which facilitate understanding.

1.1 Motivating example.

Mapping being necessarily platform-dependent, we shall consider in this paper shared memory multi-core platforms with a specific memory organization, detailed in Section 3.1. The example in Fig. 1 provides a simple dataflow program, a very simple C implementation with two threads, and the corresponding implementation model (in the middle column).

¹<https://www.mathworks.com/products/simulink.html>

²<http://www.esterel-technologies.com/products/scade-suite/>

³Sequencing of blocks into threads executed by processors; code, stack and data variables to memory locations; synchronizations to semaphores, *etc.*

```

fun Prod:()->(int)
fun Cons:(int)->()
var x:int
let
  (x) = Prod( )
  ( ) = Cons(x)
tel

```

(a)

```

fun Prod:()->(int) at 0x20100
fun Cons:(int)->() at 0x30200
var x:int at 0x22000 on cpu0
  x_cpu1:int at 0x22000 on cpu1
  x_ram:int at 0x22000
  u:event u1:event v:event v1:event
  p:event q:event r:event s:event
let
  thread on cpu0 at 0x20000 stack 0x30000
    top done(q) [wait:sem_1] (_)= (v1)
    top wait(q) (x)= Prod( )
    top done(p) [flush:0x22000] (x_ram)= (x)
    top wait(p) [signal:sem_0] (u)= top
  thread on cpu1 at 0x30000 stack 0x40000
    top done(r) [wait:sem_0] (_)= (u1)
    top wait(r) [inval:0x22000] (x_cpu1)= (x_ram)
    top done(s) ( )= Cons(x_cpu1)
    top wait(s) [signal:sem_1] (v)= top
  tel

```

```

semaphore sem_0 (u) (u1) init false on lock_0
  (u1)= (u)
semaphore sem_1 (v) (v1) init true on lock_1
  (v1)= top fby (v)

```

```

1   extern int x ;
2   extern int x_cpu1 ;
3   __attribute__((section(".oncpu0")))
4   void init() { signal(lock_1) ;}
5
6   __attribute__((section(".oncpu0")))
7   void thread_cpu0() {
8     init() ; global_barrier() ; for(;;) {
9       wait(lock_1) ;
10      Prod(&x) ;
11      flush(&x) ;
12      signal(lock_0) ;
13    }
14  __attribute__((section(".oncpu1")))
15  void thread_cpu1() {
16    global_barrier() ; for(;;) {
17      wait(lock_0) ;
18      inval(&x_cpu1) ;
19      Cons(&x_cpu1) ;
20      signal(lock_1) ;
21    }
22
23  ldscript fragment:
24    x=0x22000; x_cpu1=0x22000;
25    stack0=0x30000; stack1=0x40000;
26    .=0x20000; .bank2:{*(.oncpu0)};
27    .= 0x100 ; *(.Prod) ;
28    .=0x30000; .bank3:{*(.oncpu1)};
29    .= 0x200 ; *(.Cons) ;
30

```

(b)

(c)

Figure 1: Simple producer-consumer specification (a), two-thread implementation (c), and the corresponding implementation model (b) providing both the data-flow model of the implementation (in black) and the mapping information allowing execution on the HW platform (in red). Line numbering is common between (b) and (c).

The specification program follows a simplified Lustre [4] syntax, presented in Section 2, and defines a simple producer-consumer application with a single communication variable `x`. The C implementation is much more complex. While the production and consumption function calls can still be identified, they are surrounded by calls whose function is to ensure correct execution on the asynchronous multi-core platform:

- Semaphore `wait` and `signal` API calls ensure that production happens before consumption, and that consumption is completed when a communication variable is reused for production.
- Data cache `dflush` and `dinval` API calls implement the memory coherency protocol ensuring that the consumer uses the correct data.

Note that the multi-threaded implementation consists not only of C code, but also comprises GCC annotations and the linker script which allow the definition of the memory mapping.

Such tightly-controlled mapping is often employed in critical embedded systems. In avionics applications like our case study, the worst case execution time must be demonstrated for normal conditions, but the application must also be robust to “external factors”. The choice of a semaphore-synchronized implementation helps with the second objective, largely improving robustness by essentially guaranteeing the respect of the functional semantics. Achieving the first objective can then be done through tight control of memory allocation and synchronization, and through the use of hardware with good support for timing predictability (hardware semaphore implementation, no shared caches, *etc.*). These design choices, not covered in this paper, reduce timing variability and facilitate timing analysis.

As explained above, the implementation model of Fig. 1(b) consists in a dataflow program (in black) extended with annotations defining all the aspects of its mapping (in red). The dataflow program uses some extensions to Lustre allowing the description of synchronization. These extensions, defined in Section 2, include the synchronization data type `event` and the `wait` and `done` constructs that allow the definition of sequencing constraints not implied by data variables. The dataflow implementation program, endowed with Kahnian asynchronous semantics (defined in Section 2.2) provides a precise functional model of the execution on platform. For instance, specification variable `x` is replaced here with 3 variables `x`, `x_cpu1`, and `x_ram` allowing the representation of the various states of the memory system where the value produced on processor `cpu0` has not yet been propagated to the RAM or to the cache of processor `cpu1`. The implementation model also provides dataflow interpretations for the various API calls. For instance, in line 13 the dataflow interpretation of `dflush` ensures that the local value of `x` has been propagated onto its RAM counterpart `x_ram`, and in line 14 the dataflow interpretation of `signal` produces a token (the special literal `top`) that can be consumed later by a `wait` call. The equations in lines 25 and 27 are not part of threads. They provide the semantics of platform semaphores.

Mapping information defines the construction of sequential threads and their allocation to processors, the allocation of all code, stack and data to memory, and the implementation of interprocessor synchronization with semaphores that are allocated onto hardware locks. Note that the C code is obtained through a pretty printing of implementation model elements. For instance, there exists a line-to-line correspondence between equations of a thread in the implementation model and function calls in the infinite loop of the corresponding C thread. Also note that an explicit resource allocation is needed to allow the representation of efficient implementations. The initial source code could be given a translation into multi-threaded code, but such a translation would ignore (optimized) resource allocation issues.

Outline. The remainder of the paper will detail the elements of our implementation language, and show that it can be used to represent efficient implementations of large-scale applications.

Section 1.2 presents related work. Section 2 presents the modifications we bring to the Lustre/Scade language, introduces its less commonly known Kahnian asynchronous semantics, and presents an efficient implementation of our simple example. Section 3 defines the syntax and (platform) semantics of the implementation modeling language. Section 4 presents our experimental results, and Section 5 concludes.

1.2 Related work and originality

Most work on parallel application mapping (*e.g.* [2, 1]) involves a *code generation* phase that escapes formal analysis, covering at least some of the aspects we consider here: construction of threads, synthesis of synchronization and memory consistency protocols, memory allocation, *etc.* From this perspective, our work is an attempt at uniform formal modeling of this phase, which dissociates platform-independent aspects from platform-dependent ones, and which currently covers shared memory, semaphore-synchronized platforms.

Our formalization effort can also be seen as the first step towards formal proof of correctness for this code generation phase. This is similar to previous work on formally proved compilation [11], on static analysis of parallel software [13], and on rigorous systems design using BIP [3]. The main difference is that we remain solidly attached to a dataflow, deterministic semantic model even in the description of implementations. We expect this choice will simplify the specification and proof of correctness results.

Previous work on the compilation and the translation validation for the Signal language [10, 14] also maintains an end-to-end dataflow formalization. By comparison, our approach extends dataflow modeling to cover low-level multi-processor implementation issues (semaphore synchronization, memory consistency, construction of threads).

The objective of reducing the semantic distance between specification and implementation is also covered in [17]. The difference is the dataflow model with simpler control structure that we consider, and the fact that we consider aspects such as synchronization, memory consistency, and memory allocation.

From a more classical modeling point of view our work does not aim for the generality of UML/MARTE [12], but rather to provide a specific solution to the problem of correct multi-thread implementation of dataflow synchronous specifications. In this, it joins previous work that enriches dataflow languages with annotations describing non-functional requirements [2].

2 Modifications to Lustre

This section introduces a dataflow synchronous language for *system-level* functional specification and for defining the dataflow part of *implementation* models. We could define this language as a strict extension of Lustre [4] or Scade with new constructs for our new modeling needs. However, the system-level and implementation-oriented perspective makes some major features of these languages unneeded, and including them would only pollute our presentation. For this reason, we remove them. We shall not insist here on the syntax and semantics of Lustre, which has been covered elsewhere. Instead, we focus on the various modifications.

We call the new language InteLus, for Integration Lustre, and its syntax is provided in the black and blue parts of Fig. 2. Unlike Lustre and Scade, which also allow the programming of the sequential tasks of an embedded system, InteLus is only designed to allow the system-level integration of these tasks. For this reason, InteLus does not have a module system. One aspect of this is that a full-fledged interface definition is not needed at system level. We only identify input variables, which intuitively correspond to memory-mapped input devices.

```

<prog> ::= <type>* <fun>* var <var>* let <ceq>* <thread>+ <sem>* tel
<type> ::= type <id> size <int>
<fun> ::= fun <id> (<list(<arg>)>) -> (<list(<arg>)>) <alloc>?
<arg> ::= state? <tid>
<var> ::= input? <id> : <tid> <varalloc>?
<tid> ::= bool | int | <id> | event
<ceq> ::= <guard>? <wait>? <done>? <eq>
<eq> ::= (<nvlst(<lval>)>) = (<nvlst(<expr>)>)
          | (<list(<lval>)>) = <fid>(<list(<expr>)>)
          | (<id>) = <k> fby <expr>
          | (<id>) = <expr> when <id>
          | (<id>) = merge <id> <expr> <expr>
<expr> ::= <k> | <id>
          <k> ::= <lit> | <fid>()
<lit> ::= true | false | <int> | top
<lval> ::= <id> | _
<list(X)> ::= | <nvlst(X)>                                (list meta-rule)
<nvlst(X)> ::= X | <nvlst(X)>,X                         (non-void list meta-rule)
<wait> ::= wait(<list(<id>)>)
<done> ::= done(<list(<id>)>)
<guard> ::= top | <id> | <guard> <wait>? <done>? on <id>
<alloc> ::= at <addr>
<varalloc> ::= <alloc> <cpuid>?
<thread> ::= thread on <cpuid> <alloc> stack <addr><teq>*
              <teq> ::= <ceq> | <capi>
              <capi> ::= <guard> <wait>? <done>? <api>
              <api> ::= [wait:<id>] (<lval>) =(<id>)
                        | [signal:<id>] (<id>) = (<expr>)
                        | [dintval:<addr>] (<nvlst(<id>)>) = (<nvlst(<id>)>)
                        | [dfflush:<addr>] (<nvlst(<id>)>) = (<nvlst(<id>)>)
<sem> ::= semaphore <id> (<nvlst(<id>)>) (<nvlst(<id>)>)
          on <lockid> <eq>+

```

Figure 2: Lustre language subset (in black). InteLus extensions (blue). Extension with mapping information (red).

Furthermore, InteLus assumes that all sequential tasks have already been built, taking the form of sequential functions⁴ called from the InteLus program (like `Prod` and `Cons` in Fig. 1(a,b)). All memory used by the functions must be exposed to the memory allocation algorithms. For this reason, functions cannot contain static variables. To allow the representation of stateful tasks, the state must be exposed as one or more dataflow variables transmitted from one cycle to the next using `fby` constructs. To allow efficient memory allocation of such variables, function inputs and outputs may be tagged with the `state` keyword, to identify inputs that are passed through reference, modified in place and passed on as outputs. Inputs and outputs marked with `state` must be paired and placed at the beginning of the input and output argument lists. The following function declaration features one state variable: `fun myfun (state int, int)-> (state int, bool)`

InteLus incorporates the `event` type of Signal [10], with the difference that its only value is here `top`, and not `true`. Variables of this type carry no information, representing pure synchronization. In Fig. 1(b) we use them to provide the dataflow interpretation of semaphore operations. We also use them to define control dependencies between equations that do not exchange data. This is done through the use of the novel `wait` and `done` constructs. When placed in front of an equation, `wait(s1, ..., sk)` will delay the start of the equation until a `top` value (a token) can be read on each of the `event` type variables s_1, \dots, s_k . When placed in front of an equation, `done(s1, ..., sk)` will write `top` on each of the variables s_1, \dots, s_k after the completion of the execution of the equation.

Sequencing dataflow equations to build threads running in an asynchronous environment requires a normalization phase. For each equation of a thread, this phase builds a *guard*, which is the (possibly empty) cascade of tests needed to determine, at each execution cycle where the control reaches the equation, if the equation is executed *or not*, thus allowing giving control in sequence.⁵ Guards are placed in front of equations. A guard always starts with a variable of type `event` (or the literal `top`) which identifies the *trigger* event of the cascade of tests. We require that all equations of a thread have the same trigger. It represents the event triggering iterations of the thread body. When the trigger is `top`, no external event is needed to trigger iterations, and the thread body is an infinite loop (like in our example). Triggers different from `top` allow the representation of interrupt-driven tasks (not present in our examples).

In each guard, the trigger is followed by a sequence of tests, and synchronizations. The test of Boolean variable C is represented with “on C ”. The constructs `wait` and `done` allow the definition of control dependencies at all levels of guard decoding. Consider the following equation: `u on x wait(a) done(b) on y (z) = f(t)`

At each cycle where u is present and $x=true$, a `top` value is waited for on variable a after the test on x was performed and before the test on y is executed. During the same cycles, a `top` value is written on variable b when the execution of the equation is completed, if $y=true$, or just after the test on y , if $y=false$. Note how, unlike previous uses of guards in synchronous languages [16], our guards focus on synchronization, clearly representing the flow of control from trigger to cascade of tests and synchronizations, until passing control in sequence.

An InteLus program featuring non-trivial guards is provided in Fig. 3 (the black, dataflow part). This program is an optimized, pipelined implementation of the producer-consumer specification of Fig. 1(a). It contains several features not present in the non-optimized program of Fig. 1(b). The Boolean variable c is tested by guards to allow the incremental activation of equations during the prologue of the pipelined implementation. Software pipelining requires some memory replication, to allow `Prod` and `Cons` to work in parallel on different copies of the specification variable x . The copy used by `Prod` is located at address `0x20500`, and that used by

⁴Compiled from traditional Lustre nodes, or other C functions.

⁵The decision not to execute is particularly valuable in an asynchronous environment where absence cannot be detected.

```

1  fun Prod:()->(int) at 0x20100
2  fun Cons:(int)->() at 0x30200
3  var x:int at 0x20500 on cpu0 x1:int at 0x30500 on cpu0
4    x1_ram:int at 0x30500 x1_ram_fby:int at 0x30500
5    x2:int at 0x30500 on cpu1 x2_ram:int at 0x30500
6    c:bool at 0x30504 on cpu1
7    i:event j:event k:event l:event m:event n:event o:event p:event
8    q:event v:event v0:event v1:event u:event u1:event
9  let
10   (l) = top fby k
11   (q) = top fby p
12   (x1_ram_fby) = 0 fby x1_ram
13   (x2_ram) = x1_ram_fby when c
14  thread on cpu0 at 0x20000 stack 0x30000
15  top wait(1)          (x) = Prod()
16  top done(i)          [wait:s1]      (_) = (u1)
17  top wait(i)          (x1) = (x)
18  top done(j)          [flush:0x20504] (x1_ram) = (x1)
19  top wait(j) done(k) [signal:s0]    (v) = top
20  thread on cpu1 at 0x30000 stack 0x40000
21  top wait(q) done(m) on c [wait:s0]      (_) = (v1)
22  top wait(m)          on c [invalid:0x20504] (x2) = (x2_ram)
23  top done(n)          on c          () = Cons(x2)
24  top wait(n) done(o)  [signal:s1]      (u) = top
25  top wait(o) done(p)          (c) = false fby true
26  semaphore s0 (v) (v1) init false on lock_0
27  (v0) = top fby v
28  (v1) = v0 when c
29  semaphore s1 (u) (u1) init true on lock_1
30  (u1) = top fby u
31  tel

```

Figure 3: Efficient pipelined implementation of our example

`Cons` at address 0x30500. The copy from one location to another is realized at each cycle by the equation in line 20.

Fig. 3 also features equations not assigned to threads or semaphores. These are equations that are needed for the completion of the dataflow model, but whose semantic action will either be implemented by the sequencing of threads (for lines 13 and 14) or whose semantic action will not require encoding with platform operations. For instance, the equations in lines 15 and 16 need no platform operation due to the allocation of input and output variables on the same memory locations.

Finally, notation “`_`” as an lvalue for equation output values of type `event` whose value is not needed. This notation reduces the number of variables.

2.1 Synchronous semantics principles

The semantics of InteLus is close to that of Lustre, which is well covered in existing literature [4, 5]. For this reason, and due to space restrictions, we will provide the full synchronous semantics of InteLus in a separate report. We provide here only the principles of the formalization, used next.

The semantics of InteLus statements and programs is described through structural operational semantics (SOS) rules used to derive transitions of the form $p' \xrightarrow[I]{O} p''$. Here, p' and p'' are terms describing the input and output state of some statement or program p , I is a valuation of all input variables of the statement/program, and O is a valuation of all variables that are not inputs. The transition defines the behavior of p for one execution cycle.

State terms associated to a statement or program p describe the values currently stored by `fby` equations. They do so by modifying the constants of `fby` equations directly in the text of p . The initial state is p (with the initial `fby` constants). Values assigned by I and O to variables are either of the type of the variable, or can be the special value `nil` representing the absence of a value. If V is a set of variables, we denote with \mathcal{R}_V the set of all possible assignments of the variables in V . For $r \in \mathcal{R}_V$ and $v \in V$ we denote with $r(v) \in \text{Type}(v) \cup \{\text{nil}\}$ the value of v in r .

Given a program p , an execution trace is any sequence of transitions starting in the initial state: $p \xrightarrow[I_1]{O_1} p_1 \xrightarrow[I_2]{O_2} \dots \xrightarrow[I_n]{O_n} p_n$. We say that a program is correct if a trace exists for every sequence I_1, \dots, I_n assigning values different from `nil` to every input variable. We denote the set of all traces of program p with $\text{Traces}(p)$. Given the determinism of the synchronous semantics, $\text{Traces}(p)$ can be seen as a sub-set of \mathcal{R}_V^* , where V is the set of variables of p .

The full semantics is described in Appendix A.

2.2 Kahnian asynchronous interpretation

The determinism of the InteLus dataflow equations means that we can endow InteLus programs with **asynchronous** Kahn process networks (KPN) [7] semantics without losing determinism. This asynchronous interpretation is important in our case, given the objective of asynchronous multi-processor implementation. It puts into evidence well-formed properties allowing implementation on the target execution platforms. A strong semantics preservation property links the asynchronous interpretation of an InteLus program to its synchronous semantics, and its small-step operational description facilitates the link with the platform semantics of Section 3.3. This facilitates the formal characterization of implementation correctness.

2.2.1 Preliminary notations

In our semantics, program equations are deterministic Kahn processes communicating through infinite lossless FIFO channels corresponding to the variables. The semantics is asynchronous, so

$$\begin{array}{c}
\frac{\forall i \in \overline{1, m} : \mathbf{adv}(\mathbf{h}, Y_i, k) \quad (x_1, \dots, x_n) = F_{fid}(h_k(Y_1), \dots, h_k(Y_m))}{\ll (X_1, \dots, X_n) = fid(Y_1^k, \dots, Y_m^k), h \gg \rightarrow \ll (X_1, \dots, X_n) = fid(Y_1^{k+1}, \dots, Y_m^{k+1}), h\delta(\langle X_i \mapsto x_i \mid i = \overline{1, n} \rangle) \gg} \quad (\mathbf{fcall}) \\
\frac{}{\ll X = k \mathbf{fby} Y^n, h \gg \rightarrow \ll X = h_n(Y) \mathbf{fby} Y^{n+1}, h\delta(\langle X \mapsto h_n(Y) \rangle) \gg} \quad (\mathbf{fby}) \\
\frac{\mathbf{adv}(\mathbf{h}, C, m) \quad h_m(C) = \mathbf{false} \quad \mathbf{adv}(\mathbf{h}, X, m)}{\ll Y = X^m \text{ when } C^m, h \gg \rightarrow \ll Y = X^{m+1} \text{ when } C^{m+1}, h \gg} \quad (\mathbf{when-}) \\
\frac{\mathbf{adv}(\mathbf{h}, C, m) \quad h_m(C) = \mathbf{true} \quad \mathbf{adv}(\mathbf{h}, X, m)}{\ll Y = X^m \text{ when } C^m, h \gg \rightarrow \ll Y = X^{m+1} \text{ when } C^{m+1}, h\delta(\langle Y \mapsto h_m(X) \rangle) \gg} \quad (\mathbf{when+}) \\
\frac{\mathbf{adv}(\mathbf{h}, C, m) \quad h_m(C) = \mathbf{true} \quad \mathbf{adv}(\mathbf{h}, X, n)}{\ll Z = \mathbf{merge} C^m X^n Y^p, h \gg \rightarrow \ll Z = \mathbf{merge} C^{m+1} X^{n+1} Y^p, h\delta(\langle Z \mapsto h_n(X) \rangle) \gg} \quad (\mathbf{merge+}) \\
\frac{\mathbf{adv}(\mathbf{h}, C, m) \quad h_m(C) = \mathbf{false} \quad \mathbf{adv}(\mathbf{h}, Y, p)}{\ll Z = \mathbf{merge} C^m X^n Y^p, h \gg \rightarrow \ll Z = \mathbf{merge} C^{m+1} X^n Y^{p+1}, h\delta(\langle Z \mapsto h_p(Y) \rangle) \gg} \quad (\mathbf{merge-}) \\
\frac{\ll p, h \gg \rightarrow \ll p', h\delta(x) \gg \quad \forall i : \mathbf{adv}(\mathbf{h}, S_i, k) \quad \forall i, j : (x(S_i) = x(T_j) = \mathbf{nil}) \wedge (S_i \neq T_j)}{\ll r(p, k), h \gg \rightarrow \ll r(p', k+1), h\delta(x \cup \langle T_i \mapsto \mathbf{top} \mid i = \overline{1, n} \rangle) \gg} \quad (\mathbf{wait-done}) \\
\frac{\mathbf{adv}(\mathbf{h}, C, k) \quad h_m(C) = \mathbf{true} \quad \ll eq, h \gg \rightarrow \ll eq', h\delta(r) \gg}{\ll \mathbf{on} C^k eq, h \gg \rightarrow \ll \mathbf{on} C^{k+1} eq', h\delta(r) \gg} \quad (\mathbf{on+}) \\
\frac{\mathbf{adv}(\mathbf{h}, C, k) \quad h_m(C) = \mathbf{false} \quad (\mathbf{on-}) \quad \mathbf{adv}(\mathbf{h}, S, k) \quad \ll eq, h \gg \rightarrow \ll eq', h' \gg}{\ll \mathbf{on} C^k eq, h \gg \rightarrow \ll \mathbf{on} C^{k+1} eq', h' \gg} \quad (\mathbf{grd}) \\
\frac{}{\ll eq_i, h \gg \rightarrow \ll eq'_i, h' \gg} \\
\frac{}{\ll eq_1, \dots, eq_i, \dots, eq_n, h \gg \rightarrow \ll eq_1, \dots, eq'_i, \dots, eq_n, h' \gg} \quad (\mathbf{interleave})
\end{array}$$

Figure 4: Kahn process network semantics. Predicate $\mathbf{adv}(\mathbf{h}, X, k)$ is defined as $\text{len}(h(X)) > k$. Term $r(p, k)$ is defined as $\mathbf{wait}(S_1^k, \dots, S_m^k) \mathbf{done}(T_1, \dots, T_n) p$

that absence of a variable cannot be reacted upon (only presence). Execution traces are represented with *histories* assigning sequences of values different from \mathbf{nil} to each variable identifier. Given a set of variables V , we denote with $Hist(V)$ the set of finite histories h assigning to each $v \in V$ a sequence of values $h(v) \in Type(v)^*$. On $Hist(V)$ we introduce the concatenation operation, defined pointwise by $(h_1 h_2)(v) = h_1(v) h_2(v)$ for all v . Operator $\text{len}(x)$ provides the length of a string.

The asynchronous observation of a synchronous trace discards synchronization, retaining for each variable the sequence of values different from \mathbf{nil} . It is defined as $\delta : \mathcal{R}_V^* \rightarrow Hist(V)$ where $\delta(t_1 t_2) = \delta(t_1) \delta(t_2)$ for all $t_1, t_2 \in \mathcal{R}_V^*$ and $\delta(r)(v) = r(v)$ if $r \in \mathcal{R}_V$ with $r(v) \neq \mathbf{nil}$ and $\delta(r)(v) = \epsilon$ (the empty word), if $r(v) = \mathbf{nil}$.

2.2.2 Program state

In the classical definition of Kahn process networks [7], system state is fully determined by the state of the processes and the state of the communication channels. The state of each communication channel can be represented with the sequence of messages produced, but not yet consumed on that channel.

In InteLus, under asynchronous semantics, equations are stateless. This is true even for the \mathbf{fby} equation, whose internal state is naturally conflated with that of the output variable. On the other hand, point-to-point channels are replaced with multi-cast variables that can be read by multiple equations. For this reason, we represent states of a program or statement p as pairs $\ll p', h \gg$ where:

- $h \in Hist(V)$, where V is the set of variables of p . It gives the finite (possibly empty) sequence of values different from \mathbf{nil} that were assigned to each variable since the beginning

of the execution. If v is the output of a **fby** equation, $h(v)$ also contains, on the last position, the internal state of the **fby** (its constant).

- p' is an annotated term over p . The terms are based on those of the synchronous semantics. In addition, for each equation eq that reads a variable v , the use of v inside the equation is annotated with a non-negative integer defining the position in $h(v)$ that the equation will read at its next execution. Indices in $h(v)$ start at 0.

Note that there is some redundancy between the **fby** constants in the term and the same values stored in the history of the **fby** output variables. We keep both for consistency with the synchronous semantics. The initial state of a program/statement p is $\ll p_0, h_0 \gg$, where the p_0 is obtained from p by annotating each variable read point with 0. If v is defined by equation $v = k \text{ fby } y$ then $h_0(v) = k$. For all other variables $h_0(v) = \epsilon$.

We say that a state $\ll p, h \gg$ is complete if for every variable v defined by $v = k \text{ fby } y$ all read pointers are equal to $\text{len}(h(v)) - 1$, and for all other variable w the read pointers are equal to $\text{len}(h(w))$. Complete states can be put in direct relation with states of the synchronous semantics. We denote with $\text{strip}(\ll p, h \gg)$ the synchronous state term obtained from $\ll p, h \gg$ by removing the history h and the read point annotations.

2.2.3 Semantic rules and semantics preservation

Semantics is presented in Fig. 4, under SOS form. Concurrency is by interleaving. At each transition, exactly one equation is executed among those that can be executed (cf. rule **(interleave)**). The rules for guards (**on***) and (**grd**) are optional. Without them, the semantics works for programs without guards. With the rules, equations can have guards. Input variables are not considered in the semantics. It is assumed that each input is read by exactly one equation, and that a fresh value is provided at each execution of that equation.

Predicate $\text{adv}(h, v, n)$ determines whether we can read the n^{th} element of $h(v)$ in history h . It is used to determine if enough input is available to enable the execution of a transition.

The synchronous and Kahnian asynchronous semantics of InteLus are tightly related by the following result:

Theorem 1 (Semantics preservation) *Let p be a correct synchronous program. Then:*

- For any $r_1 \dots r_k \in \text{Traces}(p)$ there exists a complete asynchronous trace $\ll p_0, h_0 \gg \rightarrow \dots \rightarrow \ll p_n, h_n \gg$ such that: $h_n = \delta(r_1 \dots r_k) \delta(r)$ (1)

where $r(v) = \epsilon$ if v is not the output of a **fby**, and $r(v) = k$ if k is the state of the **fby** defining v in p_n . When this happens, the final state of the synchronous trace is $\text{strip}(\ll p_n, h_n \gg)$.

- For any asynchronous trace $\ll p_1, h_1 \gg \rightarrow \dots \rightarrow \ll p_m, h_m \gg$ of p there exists an extension $\ll p_1, h_1 \gg \rightarrow \dots \rightarrow \ll p_n, h_n \gg$ ($n \geq m$) that is complete, and there exists $r_1 \dots r_k \in \text{Traces}(p)$ satisfying property (1) above.

Proof sketch: Direct application of Theorem 4.4 of [15]. The tedious part of the proof consists in interpreting equations of p as micro-step state transition systems (μSTS) and proving that the synchronous (resp. asynchronous) composition of the μSTS s associated to equations faithfully represents the synchronous (resp. asynchronous) semantics of p . Once this is done, Theorem 4.4 can be applied, after noting that the synchronous correctness of the program ensures that the asynchronous composition of μSTS s is non-blocking. ■

2.2.4 Properties ensuring implementability

The Kahnian semantics is the natural semantic framework for defining a series of correctness properties ensuring the implementability of our programs. We briefly present here only 3 such properties. These properties are amenable to low-complexity verification and synthesis through the use of sufficient properties defined in the synchronous model.

Boundedness. When the program of Fig. 1(a) is executed under Kahnian semantics, the producer may be executed an indefinite number of times before the consumer is executed once. Such a behavior requires infinite storage, and is not amenable to static resource allocation. It must be prohibited through the introduction of `wait/done` dependencies. Our objective is that implementations require the storage of at most one value at a time for each variable. Semantically, this amounts to requiring that the dataflow part of the implementation program satisfies the following property:

Definition 1 (1-boundedness) *An InteLus program p is called 1-bounded if in any reachable state $\ll p', h' \gg$ of the Kahnian semantics, for every variable v that is not an input and for every read point annotation x of v , we have: $\text{len}(h'(v)) - x \leq 1$.*

Implementation of `fby` with no internal memory. The general translation of `fby` into C code requires the use of internal storage. However, all memory must be exposed to optimization, so we always impose scheduling constraints (through `wait/done` dependencies) ensuring that internal storage is not needed.⁶

Explicit synchronization. Under dataflow semantics, data accesses are synchronizing. Under machine semantics, they are not. Assume that a variable v is used for communication between equations eq_1 and eq_2 that will be mapped on different threads. Then, variables of type `event`, later mapped onto semaphore operations, must be used to represent the synchronization associated to these data accesses. The completeness of the synchronization can be checked on the dataflow implementation model. Indeed, if enough synchronization has been added, then the Kahnian semantic transitions of eq_2 do not need to perform the synchronization tests on v (the `adv` conditions in Fig. 4).

3 Implementation modeling

This section starts with a presentation of the architectures we target, including a precise description of their API and of the desired implementation structure. Based on this description, we can introduce the implementation model: its syntax, intuitive and formal semantics.

3.1 Target architectures

For the scope of this paper, we shall consider multi-core shared memory architectures without shared caches and without hardware cache coherency between L1 caches. This definition covers classical multi-core processors when their shared caches are disabled (often the case in embedded contexts), or parts of larger architectures, such as the computing tiles of the Kalray MPPA 256

⁶Even stricter rules can be imposed, to reduce memory consumption or enforce internal coding rules. In our avionics case study we ensure that the input and output variables of a `fby` can share the same memory location, which also means that no code is needed in the implementation for `fby` equations. Such stricter rules must be used with care, because they can introduce deadlocks.

many-core⁷, which we use as test platform. Due to presentation size restrictions, we do not cover here issues related to the interaction between the shared memory (sub-)system we consider and its environment (communication buses, on-chip networks, DMA units, *etc.*), or to the use of a memory management unit (MMU). We assume that our platforms have a single address space with physical addressing and no protection.

3.1.1 API and ABI

We assume that our target architectures provide a minimal set of services, and we standardize access to them by means of a simple API. Similarly, we make a number of assumptions on the way software is deployed. These assumptions can be seen as an extended application binary interface (ABI). The definition of the semantics of our implementations heavily relies on the API and ABI defined next. Modifications on any of the two may require changes to the operational semantics defined next.

Cache consistency. Software running on one CPU core can enforce consistency between its L1 data cache and the shared RAM using 2 primitives. Primitive `inval(addr)` invalidates the cache line containing address `addr`. The next access to an address in this cache line is guaranteed to be a cache miss, which forces loading from the shared RAM. Primitive `flush(addr1, ..., addrk)` forces the writing of the words at addresses $addr_1, \dots, addr_k$ to the shared RAM, and enforces a memory barrier afterwards. When the execution of `flush` completes, all the words have already been written to shared RAM.

Semaphores. Synchronization between threads is done using binary semaphores. A fixed, finite set of semaphores is provided. The state of all semaphores is initially `false`. Synchronization is done using 2 primitives. Primitive `signal(l)` should only be called when the state of semaphore l is `false` (otherwise, the behavior is undefined). It changes the state of l to `true`. Primitive `wait(l)` waits until the state of semaphore l is `true` and changes the state to `false`. When multiple `wait(l)` statements are waiting when `signal(l)` sets the semaphore to `true`, then only one of them is non-deterministically chosen and executed. Our implementations will ensure that no concurrency exists between `wait(l)` calls.

ABI We shall consider in this paper only statically scheduled, bare metal implementations. Like in our example of Fig. 1, each CPU is assigned one sequential thread – a function that never terminates. An initialization function is called by one of the threads when execution starts. A global synchronization barrier (function `global_barrier`) separates initialization from the rest of the execution. We assume the software is already deployed, meaning that we do not cover here boot-related issues. Each thread is loaded at a fixed address along its local data (if any). The same is true for functions called by threads. For each thread, the stack base is statically defined.

3.2 Mapping information

This section provides the description of the red syntax elements of Figures 2, 1, and 3, which define the mapping of the data-flow variables and equations onto resources of the target platform. Mapping annotations have 3 functions. First, they **partition the data-flow equations** in 3 classes: equations sequenced into threads, equations describing the behavior of platform devices (in our case semaphores), and equations that do not need sequencing into threads due to specific

⁷www.kalray.eu/kalray/products/#processors

$\frac{L([s]) = \text{true}}{\bullet [wait : [s]]eq, M, L \xrightarrow[p]{} [wait : [s]]eq\bullet, M, wait(L, [s])} \text{ (wait)}$ $\frac{L([s]) = \text{false}}{\bullet [signal : [s]]eq, M, L \xrightarrow[p]{} [signal : [s]]eq\bullet, M, signal(L, [s])} \text{ (sig)}$ $\frac{L([s]) = \text{true}}{\bullet [signal : [s]]eq, M, L \xrightarrow[p]{} Err} \text{ (sig-err)}$ $\bullet [dinval : adr] eq, M, L \xrightarrow[p]{} [dinval : adr] eq\bullet, dinval(M, p, adr), L \text{ (dinval)}$ $\bullet [dflush : adr] eq, M, L \xrightarrow[p]{} [dflush : adr] eq\bullet, dflush(M, p, adr), L \text{ (dflush)}$
$\frac{M([C]) = (R(_, (D \text{ true}))) \quad M([Y]) = (R(_, (D y))) \quad M([X]) = (R(0, _))}{\bullet (X) = Y \text{ when } C, M, L \xrightarrow[p]{} (X) = \bullet_{Y,C}^{X=y} Y \text{ when } C, enter(M, p, \{X\}, \{C, Y\}), L} \text{ (when+)}$ $\frac{M([C]) = (R(_, (D \text{ false})))}{\bullet (X) = Y \text{ when } C, M, L \xrightarrow[p]{} (X) = \bullet_C Y \text{ when } C, enter(M, p, \{\}, \{C\}), L} \text{ (when-)}$ $\frac{\text{conditions of (when+) or (when-) not satisfied}}{\bullet (X) = Y \text{ when } C, M, L \xrightarrow[p]{} Err} \text{ (when-err)}$ $\frac{M([C]) = (R(_, (D \text{ true}))) \quad M([Y_1]) = (R(_, (D y))) \quad M([X]) = (R(0, _))}{\bullet (X) = \text{merge } C Y_1 Y_2, M, L \xrightarrow[p]{} (X) = \bullet_{C,Y_1}^{X=y} \text{ merge } C Y_1 Y_2, enter(M, p, \{X\}, \{C, Y_1\}), L} \text{ (merge+)}$ $\frac{M([C]) = (R(_, (D \text{ false}))) \quad M([Y_2]) = (R(_, (D y))) \quad M([X]) = (R(0, _))}{\bullet (X) = \text{merge } C Y_1 Y_2, M, L \xrightarrow[p]{} (X) = \bullet_{C,Y_2}^{X=y} \text{ merge } C Y_1 Y_2, enter(M, p, \{X\}, \{C, Y_2\}), L} \text{ (merge-)}$ $\frac{\text{conditions of (merge+) or (merge-) not satisfied}}{\bullet (X) = \text{merge } C Y_1 Y_2, M, L \xrightarrow[p]{} Err} \text{ (merge-err)}$ $\frac{\text{conditions of (fcall) not satisfied}}{\bullet (X_1, \dots, X_n) = fid(Y_1, \dots, Y_m), M, L \xrightarrow[p]{} Err} \text{ (fcall-err)}$ $\frac{\forall i : M([Y_i]) = (R(_, (D y_i))) \quad \forall i : M([X_i]) = (R(0, _)) \quad (x_1, \dots, x_n) = fid(y_1, \dots, y_m)}{\bullet (X_1, \dots, X_n) = fid(Y_1, \dots, Y_m), M, L \xrightarrow[p]{} (X_1, \dots, X_n) = \bullet_{Y_1, \dots, Y_m}^{X_1=x_1, \dots, X_n=x_n} enter(M, p, \{X_1, \dots, X_n\}, \{Y_1, \dots, Y_m\}), L} \text{ (fcall)}$ $\frac{M([Y]) = (R(_, (D y))) \quad M([X]) = (R(0, _)) \quad k \neq \text{nil} \quad [X] \neq [Y]}{\bullet (X) = k \text{ fby } Y, M, L \xrightarrow[p]{} (X) = \bullet_Y^{X=y} y \text{ fby } Y, enter(M, p, \{Y\}, \{X\}), L} \text{ (fby)}$ $\frac{\text{conditions of (fby) not satisfied}}{\bullet (X) = k \text{ fby } Y, M, L \xrightarrow[p]{} Err} \text{ (fby-err)}$ $(X_1, \dots, X_n) = \bullet_{Y_1, \dots, Y_m}^{X_1=x_1, \dots, X_n=x_n} expr, M, L \xrightarrow[p]{} (X_1, \dots, X_n) = expr \bullet, exit(M, p, \{X_1, \dots, X_n\}, \{Y_1, \dots, Y_m\}), L \text{ (eq-exit)}$

Figure 5: Platform semantics, part 1. Rules for API call equations (top), rules for dataflow equations (bottom). Error rules in gray.

$\frac{M([C]) = (R(_, l)) \text{ with } l.cache(p) = (\text{D true})}{\bullet \text{on } C aeq, M, L \xrightarrow[p]{} \text{on } C \bullet aeq, M, L} \text{ (on+)}$ $\frac{M([C]) = (R(_, l)) \text{ with } l.cache(p) = (\text{D false})}{\bullet \text{on } C aeq, M, L \xrightarrow[p]{} \text{on } C aeq \bullet, M, L} \text{ (on-)}$ $\frac{\text{conditions of (on+), (on-) not satisfied}}{\bullet \text{on } C aeq, M, L \xrightarrow[p]{} Err} \text{ (on-err)} \quad \frac{aeq, M, L \xrightarrow[p]{} aeq', M', L'}{\text{on } C aeq, M, L \xrightarrow[p]{} \text{on } C aeq', M', L'} \text{ (on-comp)}$ $\frac{aeq, M, L \xrightarrow[p]{} Err}{\text{on } C aeq, M, L \xrightarrow[p]{} Err} \text{ (on-comp-err)}$ $\frac{aeq, M, L \xrightarrow[p]{} aeq', M', L'}{\text{top } aeq, M, L \xrightarrow[p]{} \text{top } aeq', M', L'} \text{ (grd-comp)} \quad \frac{aeq, M, L \xrightarrow[p]{} Err}{\text{top } aeq, M, L \xrightarrow[p]{} Err} \text{ (grd-comp-err)}$ $\bullet \text{top } aeq, M, L \xrightarrow[p]{} \text{top } \bullet aeq, M, L \text{ (grd)}$
$\frac{\text{thread } eq_1 \dots eq_k \bullet, M, L \xrightarrow[p]{} \text{thread } \bullet eq_1 \dots eq_k, M, L \text{ (seq-loop)}}{eq_i, M, L \xrightarrow[p]{} eq'_i, M', L'} \text{ (seq)}$ $\frac{\text{thread } eq_1 \dots eq_i \dots eq_k, M, L \xrightarrow[p]{} \text{thread } eq_1 \dots eq'_i \dots eq_k, M', L'}{eq_i, M, L \xrightarrow[p]{} Err} \text{ (seq-err)}$ $\frac{t_i, M, L \xrightarrow[p]{} t'_i, M', L'}{t_1 \dots t_i \dots t_k, M, L \xrightarrow[p]{} t_1 \dots t'_i \dots t_k, M', L'} \text{ (interleave)}$ $\frac{t_i, M, L \xrightarrow[p]{} Err}{t_1 \dots t_i \dots t_k, M, L \xrightarrow[p]{} Err} \text{ (interleave-err)}$

Figure 6: Platform semantics, part 2. Rules for guards (top), rules for sequential and parallel composition (bottom). Error rules in gray

mapping choices. Such are the `fby` equations satisfying the conditions of Section 2.2.4. The latter are always placed first in an implementation program, like that of Fig. 3. Partitioning annotations also include mapping information such as the allocation of threads to CPUs, the name and configuration of semaphores, *etc.*

Second, mapping annotations **define memory allocation**. In our mapping approach, these annotations cover functions, variables, and threads. For functions that are not inlined, we define the base address where both code and data are allocated. It is assumed that the entry point of the function is at that address. For threads, annotations define the addresses for code (and local data) and stack. Variables of type `event` are not allocated memory locations. All other variables are. Allocation annotations distinguish between variables viewed through different processor caches (which carry the `on` annotation) and those representing the state of the RAM.

Third, mapping annotations **provide the API implementation of various dataflow equations**.

The only platform-specific devices used in this paper are semaphores. They represent token-passing protocols where the only way to create a token is by a call to `signal`, and the only way to consume a token is by a call to `wait` (no token can be lost). It is also required that at most one token can traverse a semaphore at a given time. It can be verified that the dataflow equations of a semaphore satisfy these properties. When this happens, a platform implementation using instead the `signal` and `wait` API primitives will function correctly and preserve the dataflow semantics. The platform semantics will only consider the semaphore declaration, not the dataflow description of the protocol. This declaration includes its name, the `event` type variables that bring tokens into the semaphore, the `event` type variables that take tokens out of the semaphore, the initialization value of the semaphore, and the platform lock on which the semaphore is allocated.

3.3 Platform semantics

The platform semantics provides an operational description of the platform execution, once the application has been mapped on it and once the initialization code has been executed. It is not meant to be a semantics for general multi-threaded implementations. Instead, it only covers the very restricted control structures of the implementations we target. Furthermore, when compared to classical cycle-accurate simulation semantics, it already includes elements of abstraction meant to hide timing aspects⁸ and to isolate the semantics of the well-formed code synthesized from the implementation model from the semantics of the potentially more general C code of external functions (such as `Prod` or `Cons` in our example). This isolation means that we cannot know the exact position of control (the program counter) during execution of a function, nor the way the function manipulates data during computations. For space reasons, the platform semantics presented here covers only dataflow programs where the guard triggers are `top`.

3.3.1 State representation

In the platform semantics, states provide an abstract view of the state of the execution platform components, including CPUs, memory hierarchy, and semaphore status. They have the following structure (for clarity, we use an OCaml syntax for its definition):

⁸This form of abstraction was also employed in the concrete semantics of [13].

```

type mach_state = Err
    | Norm of ctrl_state*mem_state*lock_state
type mem_state = addr -> mem_loc_state
type mem_loc_state = W | R of posint*loc_value ;
type loc_value = { ram:value; cache:processor -> value; }
type value = D of word | U
type lock_state = lock -> bool

```

The remainder of this section details the 3 components of the machine state and defines state manipulation functions.

Control flow state. It is represented through annotation of the implementation program. The program counters of each CPU are represented with bullets (•) placed in the body of threads. To allow the identification of all possible memory access or synchronization interferences between threads, bullets can take the following positions:

- Between guarded equations of the thread, to represent the state where the execution of one equation is finished, but the next (including possible guard tests) has not yet started. When control loops back after the execution of the last equation of a thread, the bullet is first placed after the last equation, and then before the first equation of the thread.
- Before a guard test “on C”, to represent the state where the execution has reached, but not executed, the associated variable test.
- After all guard and synchronization annotations, to represent the state where the guard has been successfully traversed, but the equation execution has not started.
- Inside the equation, to represent the state where its execution has started, but is not completed. When inside an equation, the token is annotated with the memory locations corresponding to variables in use by the equation execution (read or written). These values are annotated by the transition entering the equation, and removed by the equation exit transition.

In the initial state, in each thread, the bullet is placed before the first equation. The set of all possible such state representations obtained by bullet annotations is denoted `ctrl_state`.

Memory system state. It includes the state of the RAM and that of the L1 data caches of the CPUs. We denote with `addr` the set of memory addresses that can be used by data-flow variables. At each instant, a memory location can be either written by some equation (as an output variable), or read by zero or more equations, as an input variable. Performing a read access on a variable that is currently written by another equation, or a write access on a variable that is currently written or read by another equation is an error. In the definition of type `mem_loc_state`, variant `W` corresponds to the write state, and `posint` gives the non-negative number of readers in the read access case.

In the case where a location is read, its value can be undefined (`U`) or defined (`D`) with a `word` value. The `U` value represents a state where the value of the memory location is unknown.

In a given state, read accesses from different processors to the same memory location may produce different values. To allow reasoning on this memory consistency issue, we need to store, for each memory location, not one, but `proc + 1` values, where `proc` is the number of CPU cores. Type `loc_value` structures the storage of these `proc + 1` values. The value stored on record field `ram` is that stored in RAM. The value stored on `cache(p)` is the one that the an access from CPU core `p` to its L1 cache would return.

The initial memory state is determined by the initial value of `fby` equations of the program: All locations are in state $(R(0, v))$, where $v.cache(p) = U$ for all p . Memory locations not corresponding to `fby` output variables also have have $v.ram = U$. For the output of a `fby` equation initialized with d , the initial state sets $v.ram = (D d)$.

The semantics updates memory state using 4 functions whose prototypes are given below (in OCaml syntax). The first two correspond to the `dflush` and `dinval` API calls. The remaining 2 implement the semantic actions taken upon entering and exiting an equation.

```

dinval: mem_state*processor*addr -> mem_state
dflush: mem_state*processor*addr -> mem_state
enter : mem_state*processor*(addr list)*(addr list)
       -> mem_state
exit   : mem_state*processor*((addr*word) list)*(addr list)
       -> mem_state

```

We call `dinval` ($M, p, addr$), resp. `dflush` ($M, p, addr$), upon execution of the corresponding API primitive in memory state M , on processor p , and with argument $addr$. The functions can only be applied when $M(addr) = (R(n, v))$. Their action changes v as follows: `dinval` ($m, p, addr$) sets $v.cache(p)$ to $v.ram$; `dflush` ($M, p, addr$) sets $v.ram$ to $v.cache(p)$ and $v.cache(q)$ to U for all $q \neq p$.

We denote with $[X]$ the memory location of variable X . Function `enter` ($M, p, \{X_1, \dots, X_n\}, \{Y_1, \dots, Y_m\}$) is called when starting the execution of an equation that reads variables Y_1, \dots, Y_m and writes variables X_1, \dots, X_n . The call sets the memory state of the locations $[X_i]$ to W , and increments the read counter of the locations $[Y_i]$ whenever $[Y_i]$ is not also the location of an output variable X_j .

Function `exit` ($M, p, \{(X_1, x_1), \dots, (X_n, x_n)\}, \{Y_1, \dots, Y_m\}$) is called when completing the execution of an equation that reads variables Y_1, \dots, Y_m and writes variables X_1, \dots, X_n , when the final values for the written variables are respectively $x_i, i = \overline{1, n}$. The call decrements the read counter of the locations $[Y_i]$ whenever $[Y_i]$ is not also the location of an output variable. For each of the locations $[X_i]$, it sets its state to $(R(0, v))$ where $v.cache(p) = (D x_i)$, $v.cache(q) = U$ for all $q \neq p$, and $v.ram = U$.

For simplicity, like in the Kahnian case, platform semantics does not fully consider input variables. Each input variable is given a memory location different from all other variables, it is read by only one equation, and its value is assumed ready upon reading.

Semaphore state. The binary semaphores provided by the execution platform are called *locks*. Their state is represented with Boolean values. We update `lock_state` objects using 2 functions corresponding to the `wait` and `signal` API calls:

```

signal: lock_state*lock      -> lock_state
wait  : lock_state*lock      -> lock_state

```

Function `signal` (L, l) can be called only when $L(l) = \text{false}$. It sets $L(l)$ to `true`. Conversely, `wait` (L, l) can be called only when $L(l) = \text{true}$. It sets $L(l)$ to `false`.

3.3.2 Semantic rules

Platform semantics is also provided under SOS form. Transitions have the form $s_1 \xrightarrow[p]{} s_2$, where s_1 and s_2 are objects of type `mach_state` and p is the processor performing the state transition (recall that it is an interleaving semantics). Figures 5 and 6 provides the SOS rules. States $s_i = \text{Norm}(\text{ctrl}, \text{mem}, \text{locks})$ are represented here as `ctrl, mem, locks`.

3.3.3 Correctness and semantics preservation

An implementation program is correct in the platform semantics when it satisfies two properties: it does not block or enter `Err` state, regardless of the values read from input variables, and its execution under platform semantics preserves the semantics of its dataflow part under Kahnian semantics. Semantics preservation is defined as the preservation of the sequences of values taken as input (as guards or actual equation inputs) and produced as output by the execution of the various equations (which can be represented with histories, introduced in Section 2.2.2).

To be correct, an implementation program must satisfy the properties mentioned in Section 2.2.4 and, in addition, must satisfy properties ensuring the correct use of the (sequential) resources of the platform (not presented here for lack of space).

4 Experimental evaluation

This paper introduces a language, not a mapping tool. From this perspective, we have already provided the elements showing that we can model implementations featuring optimized resource allocation. As a supplementary element of proof, this section shows that our language allows the representation of the optimized implementation of a real-life case study, produced by an automatic mapping tool.

The case study is a complex piece of critical avionics software, including more than 6000 unique dataflow nodes and 36000 variables. Its current implementation is sequential and time-triggered. Periodically, a sequence of 24 non-preemptive sequential tasks (built out of nodes) are triggered at fixed time intervals. Our first objective was to parallelize each of the 24 tasks, taken separately, *and demonstrate the correctness of the implementations*. Each task has been transformed into an InteLus specification and an automatic mapping tool performed allocation, scheduling, memory allocation, and the synthesis of the synchronization and memory coherency protocols. Implementation programs were synthesized, from which C/ldscript code allowing compilation was generated. Using the results of this paper, *the correctness of implementation can be formulated*.

The following table shows various characteristics of the largest 6 tasks (in number of nodes) and averages on the 24 tasks. The first important figure concerns memory allocation (Mem), where reuse allows a 71% reduction over the number of specification variables. We exclude input variables from this table, because they are currently excluded from optimization (work in progress).

Task	Specification		Implementation			
	Nodes	Vars*	Mem*	Sema	Locks	Speed-up
T1	1511	6077	1951 (32.1%)	1040	96 (9.2%)	10.45x
T17	1115	5008	1603 (32%)	958	108 (11%)	9.85x
T9	1090	4813	1537 (31.9%)	907	120 (13%)	11.72x
T8	1289	5239	1351 (25.8%)	1291	115 (8.9%)	13.28x
T24	1313	5894	1294 (22%)	1370	118 (8.6%)	12.49x
T16	1258	4945	1247 (25.2%)	1296	118 (9.1%)	12.97x
Avg.			28.8%		10%	10.52x

*Input variables not included.

The number of semaphores (Sema) representing point-to-point thread synchronizations is large, because very fine grain synchronization is used to allow efficient resource allocation. However, the cost of these synchronizations is very low (a few CPU cycles each) and, most important, very efficient reuse of the 127 hardware locks of the platform allows the implementation of the

semaphores. The speed-up figures are produced by an off-line scheduling model not taking into account memory interferences, and thus are not safe. However, they provide insight into application structure, and our implementation method should preserve functional semantics regardless of timing imprecision.

5 Conclusion

We have provided proof of our initial claim: Dataflow specifications can be given efficient multi-threaded implementations that preserve an internal dataflow structure. The dataflow structure of such implementations can be exposed using new language constructs. Doing so facilitates formulating the correctness of implementations, and also proving it, through the use of results such as Theorem 1, or through the use of analysis techniques specific to the dataflow domain.

Work will continue along two axes. We will enrich the implementation modeling language to incorporate new features. The first steps will be to include real-time capabilities and to diversify the target architectures by considering communication devices (DMAs, networks, external RAMs, *etc.*). The second objective is to develop a formally proven translation validation tool covering the transformation of functional specifications into implementation programs. This tool needs to be complemented later with the validation of the translation from implementation programs to code running on the platform, and with the validation of the translation from Lustre to InteLus.

Acknowledgments

The authors would like to thank Xavier Leroy, François Irigoin, and Jean Souyris for having provided significant feedback on early versions of this work.

A Synchronous semantics of InteLus

To simplify the definition of the semantics, and without affecting generality, we shall assume that literals only appear as the first argument of a **fby** statement, that no identity equations “ $x=y$ ” exist, nor “ $_$ ” lvalues, and that the input and output variables of a **fby** statement are different. This can be achieved through the use of constant and identity functions. We also make the hypothesis that each equation has (possibly empty) **wait** and **done** constructs.

A.1 State representation

The **fby** statements are the state elements of a InteLus program, each of them holding either last value of the input variable (if any) or the initial constant of the **fby** statement. The current state of a program can be represented by replacing the constants of the **fby** statements by their current state. If we consider the fragment on the left:

$$\begin{array}{ll} (y) = \text{add}(x, x) & (y) = \text{add}(x, x) \\ (z) = 10 \text{ fby } y & (z) = 6 \text{ fby } y \end{array}$$

then its initial state is the fragment itself. After one cycle where x has value 3, the new state is the fragment on the right. Note that the all program states are identical upto a constant change in **fby** equations.

A.2 Notations

Recall, from Section 2.1, that \mathcal{R}_V is the set of all mappings r assigning to each $v \in V$ a value $r(v) \in \text{Type}(v) \cup \{\text{nil}\}$. We extend this notation. We denote with $\overline{\mathcal{R}}_V$ the set of partial valuations over V . Given $r \in \mathcal{R}_V$ we define its support $\text{supp}(r)$ as the set of identifiers to which r assigns a value (possibly **nil**). We also define \bar{r} as the (partial) mapping that retains of r only the valuations different from **nil**. Note that $\mathcal{R}_V \subset \overline{\mathcal{R}}_V$.

We introduce a simple notation for (partial) mappings from variable identifiers to variable values. We denote with $\langle X_i \mapsto x_i \mid i = \overline{1, n} \rangle$ or $\langle X_1 \mapsto x_1, \dots, X_n \mapsto x_n \rangle$ a (partial) mapping assigning value x_i to identifier X_i , $i = \overline{1, n}$. The empty mapping is denoted $\langle \rangle$. The notation is ambiguous, as the domain of the mappings is not explicit.

If $I \subseteq V$ and $r \in \overline{\mathcal{R}}_V$, then $r|_I \in \overline{\mathcal{R}}_I$ is the restriction of r to the identifier set I . Note that if $r \in \mathcal{R}_V$, then $r|_I \in \mathcal{R}_I$. If $I \subseteq V$, then $\overline{\mathcal{R}}_I$ is naturally injected in $\overline{\mathcal{R}}_V$. By abuse of notation, we shall say that $\overline{\mathcal{R}}_I$ is included in $\overline{\mathcal{R}}_V$.

If $r_1, r_2 \in \overline{\mathcal{R}}_V$ such that all the bindings of r_1 are also bindings of r_2 , then we write $r_1 \leq r_2$. Relation \leq is a partial order on $\overline{\mathcal{R}}_V$. We denote with \cup the partially defined least upper bound operator associated with \leq on $\overline{\mathcal{R}}_V$. The completely defined greatest lower bound operator is denoted \cap . If $r \in \overline{\mathcal{R}}_V$ and $I \subseteq V$, then we define $r \setminus I = \langle X \mapsto x \in r \mid X \notin I \rangle$.

A.3 Structural operational semantics rules

We present the behavior of statements, code fragments, and full programs under the form of transitions defining the new state and variables assigned in one cycle for given input state and input variable assignments:

$$eqs \xrightarrow[I]{\mathcal{Q}} eqs'$$

Here, code fragments eqs and eqs' define the input and result program states, obtained by changes in the **fby** constants. If I is the set of variables taken as input by eqs , then $I \in \mathcal{R}_I$

$$\begin{array}{c}
\frac{x \neq \text{nil}}{Z = \text{merge } C X Y \xrightarrow[\langle C \mapsto \text{true}, X \mapsto x, Y \mapsto \text{nil} \rangle]{\langle Z \mapsto x \rangle} Z = \text{merge } C X Y} \quad (\text{merge+}) \\
\frac{y \neq \text{nil}}{Z = \text{merge } C X Y \xrightarrow[\langle C \mapsto \text{false}, X \mapsto \text{nil}, Y \mapsto y \rangle]{\langle Z \mapsto y \rangle} Z = \text{merge } C X Y} \quad (\text{merge-}) \\
Z = \text{merge } C X Y \xrightarrow[\langle C \mapsto \text{nil}, X \mapsto \text{nil}, Y \mapsto \text{nil} \rangle]{\langle Z \mapsto \text{nil} \rangle} Z = \text{merge } C X Y \quad (\text{merge-nil}) \\
\frac{x \neq \text{nil}}{Y = X \text{ when } C \xrightarrow[\langle C \mapsto \text{true}, X \mapsto x \rangle]{\langle Y \mapsto x \rangle} Y = X \text{ when } C} \quad (\text{when+}) \\
\frac{x \neq \text{nil}}{Y = X \text{ when } C \xrightarrow[\langle C \mapsto \text{false}, X \mapsto x \rangle]{\langle Y \mapsto \text{nil} \rangle} Y = X \text{ when } C} \quad (\text{when-}) \\
Y = X \text{ when } C \xrightarrow[\langle C \mapsto \text{nil}, X \mapsto \text{nil} \rangle]{\langle Y \mapsto \text{nil} \rangle} Y = X \text{ when } C \quad (\text{when-nil}) \\
\frac{(y_1, \dots, y_m) = F_{\text{fid}}(x_1, \dots, x_n)}{(Y_1, \dots, Y_m) = \text{fid}(X_1, \dots, X_n) \xrightarrow[\langle X_i \mapsto x_i | i = 1, n \rangle]{\langle Y_i \mapsto y_i | i = 1, m \rangle} (Y_1, \dots, Y_m) = \text{fid}(X_1, \dots, X_n)} \quad (\text{fun}) \\
(Y_1, \dots, Y_l) = \text{fid}(X_1, \dots, X_k) \xrightarrow[\langle X_i \mapsto \text{nil} | i = 1, n \rangle]{\langle Y_i \mapsto \text{nil} | i = 1, m \rangle} (Y_1, \dots, Y_l) = \text{fid}(X_1, \dots, X_k) \quad (\text{fun-nil}) \\
(Y) = k \text{ fby } X \xrightarrow[\langle X \mapsto k' \rangle]{\langle Y \mapsto k \rangle} (Y) = k' \text{ fby } X \quad (\text{fby}) \quad (Y) = k \text{ fby } X \xrightarrow[\langle X \mapsto \text{nil} \rangle]{\langle Y \mapsto \text{nil} \rangle} (Y) = k \text{ fby } X \quad (\text{fby-nil}) \\
\frac{p \xrightarrow[I]{O} p' \quad \overline{I \cup O} \neq \langle \rangle \quad \forall i, j : S_i, T_j \notin \text{supp}(I \cup O) \quad \forall i, j : S_i \neq T_j}{\text{wait}(S_1, \dots, S_k) \text{ done}(T_1, \dots, T_l) \ p \xrightarrow[I \cup \langle S_i \mapsto \text{top} | i = 1, k \rangle]{O \cup \langle T_i \mapsto \text{top} | i = 1, l \rangle} \text{wait}(S_1, \dots, S_k) \text{ done}(T_1, \dots, T_l) \ p'} \quad (\text{wait-done}) \\
\frac{p \xrightarrow[I]{O} p \quad \overline{I \cup O} = \langle \rangle \quad \forall i, j : S_i, T_j \notin \text{supp}(I \cup O) \quad \forall i, j : S_i \neq T_j}{\text{wait}(S_1, \dots, S_k) \text{ done}(T_1, \dots, T_l) \ p \xrightarrow[I \cup \langle S_i \mapsto \text{nil} | i = 1, k \rangle]{O \cup \langle T_i \mapsto \text{nil} | i = 1, l \rangle} \text{wait}(S_1, \dots, S_k) \text{ done}(T_1, \dots, T_l) \ p} \quad (\text{wait-done-nil}) \\
\frac{p_i \xrightarrow[I_i]{O_i} p'_i, i = 1, 2 \quad \text{supp}(O_1) \cap \text{supp}(O_2) = \emptyset \quad \text{supp}(O_2) \cap \text{supp}(I_1) = \emptyset}{\text{shuffle}(p_1, p_2) \xrightarrow[I_1 \cup (I_2 \setminus \text{supp}(O_1))]{O_1 \cup O_2} \text{shuffle}(p'_1, p'_2)} \quad (\text{compose-acyclic}) \\
\frac{p_i \xrightarrow[I_i]{O_i} p'_i, i = 1, 2 \quad \text{strip}(p_2) = "Y = k \text{ fby } X" \quad \text{supp}(O_1), \text{supp}(O_2), \{Y\} \text{ mutually disjoint} \quad \text{supp}(O_2) \cap \text{supp}(I_1) = \emptyset \quad Y \mapsto k' \in I_1}{\text{shuffle}(p_1, p_2) \xrightarrow[I_1 \cup (I_2 \setminus \text{supp}(O_1))]{O_1 \cup O_2 \cup \langle Y \mapsto k' \rangle} \text{shuffle}(p'_1, p'_2)} \quad (\text{compose-fby}) \\
\frac{\text{body}(p) \xrightarrow[I]{O} \text{body}(p') \quad (I \cup O) \text{ maximal in the sense of } \leq}{p \xrightarrow[I]{O} p'} \quad (\text{program-closure})
\end{array}$$

Figure 7: Operational semantics of InteLus' subset without guards. Each rule implicitly requires that the support of input and output in the inferred transition are exclusive. The *shuffle* operator takes in two equation lists and can output any equation list obtained by shuffling the two initial ones.

with $\text{supp}(I) = \mathcal{I}$. Similarly, if \mathcal{O} is the set of variables assigned by eqs , then $O \in \mathcal{R}_{\mathcal{O}}$ with $\text{supp}(O) = \mathcal{O}$. Causal correctness requires that $\mathcal{I} \cap \mathcal{O} = \emptyset$.

The structural operational semantics rules allowing to derive transitions are provided in Fig. 7. Note that, according to rule **(program-closure)**, program transitions are transitions of the program body with maximal activity, which amounts to an ASAP (as soon as possible) scheduling policy. To understand the importance of this rule, consider the following input-less program, which encodes a counter that increments x by 1 at each cycle.

```
var x:word y:word
let
  x = 1 fby y
  y = add(x,1)
tel
```

Semantic rules *(fun-nil)* and *(compose-fby-nil)* allow the construction of a program body transition where both variables are assigned value `nil`. However, rule *(program-closure)* forbids it, forcing the counter to advance at each cycle.

A.4 Clocks

In synchronous languages, *clocks* are used to represent activation conditions. Clocks represent sets of execution cycles, and can be seen as *predicates over the state and inputs of the program*.

Clocks can be compared. We say that ck_1 and ck_2 are equal, denoted $ck_1 = ck_2$ if they represent the same execution cycles, *i.e.* if the associated predicates are true at the same execution cycles. We say that ck_1 is a sub-clock of ck_2 , denoted $ck_1 \leq ck_2$ if whenever the predicate of ck_1 is true in a cycle, the predicate of ck_2 is true, too. The least upper bound and greatest lower bound are fully defined and denoted \vee and \wedge , respectively. The empty clock is denoted \emptyset , and the clock that is always present, known as the *base clock*, is denoted $.$ in formulas.

Typically, clocks are used to represent cycles where some variable is present, or where some function is executed. We denote with $clk(v)$ the clock representing the cycles where variable v is present. Consider, for instance, the following code fragment:

```
(a,b) = f(y)
(x) = y when c
(z) = y whennot c
```

The following clock relations will hold:

- $clk(a) = clk(b) = clk(y) = clk(c)$
- $clk(x) \leq clk(c)$ and $clk(z) \leq clk(c)$.
- $clk(x) \wedge clk(z) = \emptyset$ and $clk(x) \vee clk(z) = clk(y)$
- $clk(z) = clk(y) \wedge c$, where c is the current value of variable c .

Such relations are determined through static program analysis, known as *clock calculus*, and used to determine the consistency of the constraints imposed by program equations, thus proving program correctness.

$$\begin{array}{c}
\frac{eq \xrightarrow[I]{O} eq' \quad \overline{I \cup O} \neq \langle \rangle \quad \forall i, j : C, s_i, t_j \notin supp(I \cup O) \quad \forall i, j : C \neq s_i \neq t_j \neq C}{r(eq) \xrightarrow[\substack{O \cup \langle s_i \mapsto \text{top} | i = \overline{1, k} \rangle \\ I \cup \langle C \mapsto \text{true} \rangle \cup \langle t_i \mapsto \text{top} | i = \overline{1, l} \rangle}]{} r(eq')} \quad (\text{on+}) \\
\\
\frac{eq \xrightarrow[I]{O} eq \quad \overline{I \cup O} = \langle \rangle \quad \forall i, j : C, s_i, t_j \notin supp(I \cup O) \quad \forall i, j : C \neq s_i \neq t_j \neq C}{r(eq) \xrightarrow[\substack{O \cup \langle s_i \mapsto \text{top} | i = \overline{1, k} \rangle \\ I \cup \langle C \mapsto \text{false} \rangle \cup \langle t_i \mapsto \text{top} | i = \overline{1, l} \rangle}]{} r(eq)} \quad (\text{on-}) \\
\\
\frac{eq \xrightarrow[I]{O} eq \quad \overline{I \cup O} = \langle \rangle \quad \forall i, j : C, s_i, t_j \notin supp(I \cup O) \quad \forall i, j : C \neq s_i \neq t_j \neq C}{r(eq) \xrightarrow[\substack{O \cup \langle s_i \mapsto \text{nil} | i = \overline{1, k} \rangle \\ I \cup \langle C \mapsto \text{nil} \rangle \cup \langle t_i \mapsto \text{nil} | i = \overline{1, l} \rangle}]{} r(eq)} \quad (\text{on-nil}) \\
\\
\frac{eq \xrightarrow[I]{O} eq' \quad \overline{I \cup O} \neq \langle \rangle}{s \ eq \xrightarrow[\substack{O \\ I \cup \langle s \mapsto \text{top} \rangle}]{} s \ eq'} \quad (\text{trig-comp}) \\
\\
\frac{eq \xrightarrow[I]{O} eq \quad \overline{I \cup O} = \langle \rangle}{s \ eq \xrightarrow[\substack{O \\ I \cup \langle s \mapsto \text{nil} \rangle}]{} s \ eq} \quad (\text{trig-comp-nil}) \\
\\
\frac{eq \xrightarrow[I]{O} eq' \quad \overline{I \cup O} \neq \langle \rangle}{\text{top } eq \xrightarrow[I]{O} \text{top } eq'} \quad (\text{trig-top})
\end{array}$$

Figure 8: Operational semantics of guards. Rule *(trig-top* makes reaction to absence (the nil rule) unneeded. To facilitate rule writing, we denote $r(eq) = \text{"wait}(t_1, \dots, t_l) \text{ done}(s_1, \dots, s_k) \text{ on } C \text{ eq"}$

A.5 Equation guards

While clocks provide a logical representation of activation conditions, used for analysis, we also need an operational representation for them, used to generate the activation code of the implementation. The constructions used for this purpose are the guards introduced by the syntax of Fig. 2.

The semantics of the full InteLus language (with guards) is provided as an extension of that of Fig. 7 with the new rules of Fig. 8. Existing rules remain unchanged.

Under the extended semantics, if an equation has a guard, and if the trigger is present in a cycle, then the decision to execute the equation is taken by the evaluation of (present) guard variables (if any), and not by the occurrence (or not) of a present/absent variable configuration (as it was the case in the semantics without guards). In particular, when the trigger is `top`, no absence decision is needed in the execution of the equation.

When all equations of a program have a guard starting with trigger `top`, rule (**program-closure**) is no longer needed. This remains true when every maximal connected part of the dataflow contains at least one equation triggered by `top`.

Note that, according to the extended semantics, different guards may represent the same logical activation condition (the same clock), but a different causality. Consider the following code fragment:

```
top (b) = merge a b1 b2
top (c) = and(a,b)
top on a on b1 (x) = f(y)
top on c      (x) = f(y)
```

Here, the last two equations can be equivalently used to define `x`, as the same value is computed at the same cycles (on the same clock). However, the two equations do not carry the same data dependencies due to the way the control (clocks) is computed.

References

- [1] The SchedMCore software framework. <http://sites.onera.fr/schedmcore/>.
- [2] The SynDEX methodology and mapping tool. www.syndex.org.
- [3] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.H. Nguyen, and J. Sifakis. Rigorous component-based system design using the BIP framework. *IEEE Software*, 28(3), 2011.
- [4] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for programming synchronous systems. In *Proceedings POPL*, 1987.
- [5] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Proceedings ICFP*, 1996.
- [6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufman, 2008.
- [7] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, 1974.
- [8] P. Koopman. A case study of Toyota unintended acceleration and software safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf, September 2014.

- [9] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.
- [10] P. LeGuernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9), 1991.
- [11] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [12] F. Mallet and R. de Simone. MARTE: A profile for RT/E systems modeling, analysis. In *Proceedings Simutools*, 2008.
- [13] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), March 2012. <https://arxiv.org/abs/1203.3724>.
- [14] Van-Chan Ngo, J.-P. Talpin, T. Gautier, L. Besnard, and P. Le Guernic. Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools. In *Proceedings SCOPES*, 2015.
- [15] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. *Fundamenta Informaticae*, CXXVII:1–27, 2006.
- [16] J.P. Talpin, J. Brandt, M. Gemünden, K. Schneider, and S.K. Shukla. Constructive polychronous systems. *Sci. Comput. Program.*, 96:377–394, 2014.
- [17] R. von Hanxleden, B. Duderstadt, C. Motika, S. Smyth, M. Mendler, J. Aguado, S. Mercer, and O. O’Brien. SCCharts: Sequentially constructive statecharts for safety-critical applications: HW/SW-synthesis for a conservative extension of synchronous statecharts. In *Proceedings PLDI*, 2014.



**RESEARCH CENTRE
PARIS – ROCQUENCOURT**

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399